

# Stateless: an Optic Algebra Library

Jesús López-González  
j.lopezgo@alumnos.urjc.es

Juan Manuel Serrano  
juanmanuel.serrano@urjc.es

June 5, 2017

## Abstract

Abstract goes here!

## 1 Introduction

Software industry as we know it lacks well-founded frameworks to program reactive systems in a modular way. This kind of systems hide an inner *state* to their environment, showing therefore a black-box behaviour. Take a university information system as a guiding example. Universities are structured in departments, and each department is managed by a director. Even this simple definition does contain three kinds of entities: universities, departments and directors. Ideally, programming the evolving state of the whole system should be achieved by programming the state of each entity or subsystem, and finally composing them all.

In addition, composition should contemplate heterogeneity: departments could be stored in a local database while directors could be deployed as external microservices. Ultimately, any configuration should be supported, but it is essential for the system description to remain pure and decoupled from low-level details, irrelevant to the domain. Programming a declarative and modular system in this way is still a great challenge.

There are plenty of technologies to develop reactive systems, but they are postulated as solutions to very specific problems, and consequently, difficult to be reused. Turning to the university example, we could have decided to program department evolution on a relational database, like *MySQL*. However, if large volumes of data arose in time, migrating to a distributed database such as *Redis* might become necessary. Given that situation, we would find it extremely difficult to adapt existing encodings to the new setting.

Fortunately, industry has detected this issue and new abstract tools have emerged to fill the gap. In this sense, we could have avoided the aforementioned problem by programming department evolution with *Slick*, a Functional-Relational Mapper that provides a common language which can be later interpreted into different databases. Similarly, we could find other situations where a common language could unify different state evolutions. Take *optics* as an example. They could be seen as a language to achieve complex transformations over immutable in-memory data. In this scenario, *Traversal* could be used as the glue to keep university and department related.

We claim that many of these languages to access state could be further factorized, by providing a new language to unify them. Moreover, we claim that optic concepts are ideal to abstract technology-specific details away. However, prior to that, they have to be taken to a more general setting: *optic algebras*.

An optic algebra represents the essence of the state transformation associated to that optic. Generally speaking, optics were conceived to tackle a very specific problem, achieving complex

transformations over immutable in-memory data structures. For instance, a lens knows how to transform certain data structure —the focus— which is contextualized in a larger one —the whole. Regarding our university, this abstraction could perfectly describe the existing relationship between a department and its inner director. Having said so, what do we mean by the essence of a lens?

The essence of a lens requires removing all those aspects which are specific to in-memory transformations. If we take into account the classic definition of a lens:

```
case class Lens[S, A](get: S => A, set: A => S => S)
```

one could easily identify several accesses to the in-memory state  $S$ . If we abstract away from those aspects, ie. if we conceal the state, we end up with the optic algebra associated to lens:

```
case class LensAlg[F[_], A](get: F[A], set: A => F[Unit])
```

This algebra determines the existence of a focus that could be either read or updated, but it keeps itself unaware of the details underneath  $F$ . Similar derivations could be carried out over other optics, resulting in new optic algebras.

Composing optic algebras —as they were introduced in the previous paragraph— is not straightforward at all. In fact, the situation is very similar to the one that emerges when trying to compose optics in their classic representation. In this sense, Van Laarhoven and recently Profunctor optics have been postulated as alternative representations to improve composition. Taking them as inspiration, we created a new representation for optic algebras. Since it heavily relies on natural transformations, we refer to it as *natural* representation. This representation not only improves the composition up, but also introduces the notion of heterogeneity, as we will see in further sections.

Optic algebra instances for the most widespread deployments should be provided. Given this scenario, by defining our system in terms of optic algebras, we should be able to get our logic running on a database, in immutable memory, behind a microservice, or even as a combination of many of them. It is worth mentioning that generally these instantiations are not going to be as optimal as an *ad hoc* solution, but they may be good enough in most cases.

We claim that optic algebras in their natural representation lead to modular software designs. For instance, a traversal algebra could establish the connection between the university and each department. On the other hand, a lens algebra could describe the relationship between departments and directors. Then, we could compose both algebras to conform the whole system. Finally, optic algebras will enable us to do it in an heterogeneous way, with many free instances ready to go.

- 2 Background
- 3 Deriving Optic Algebras
- 4 Natural Composition and Heterogeneity
- 5 Instances for Free
- 6 Conclusions
- 7 Acknowledgements